LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-590233

Rice ROSE Compositional Analysis and Transformation Framework (R2CAT)

J. Zhao, M. G. Burke, V. Sarkar

October 10, 2012

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

Rice ROSE Compositional Analysis and Transformation Framework (R2CAT)

Jisheng Zhao, Michael G. Burke, Vivek Sarkar

Department of Computer Science, Rice University
{jisheng.zhao, mgb2, vsarkar}@rice.edu

1 R2CAT Framework

1.1 Overview

The Rice ROSE Compositional Analysis and Transformation Framework (R2CAT) synthesizes a series of program analysis and transformation components into a unified framework, and provides an interface to facilitate exchanging information among those components.

The framework is being built upon the ROSE compiler [7] which is a mature source-to-source compilation system. Some of teh distinguishing characteristics of R2CAT include:

- Sparse program analyses based on Static Single Assignment (SSA) form for scalars, arrays, and pointer data structures
- High level program analysis, i.e. based on an an Abstract Syntax Tree (AST) level program representation;
- Transformations based on incremental re-analysis.

Figure 1 gives a high level view of R2CAT. R2CAT works with the ROSE Sage Intermediate Representation (Sage IR), which is an AST-based IR. The ROSE front end parses the input programming language (e.g. C/C++) into Sage IR. The ROSE middle end constructs per-procedure Control Flow Graphs (CFGs) and the Call Graph (CG) based on the Sage IR. R2CAT interacts with ROSE to perform dataflow analyses and transformations.

Figure 2 presents the internal workflow of R2CAT, which consists of a sequence of analysis and transformation passes. A central data structure for R2CAT is the Heap SSA form program representation, which is constructed by a Heap SSA builder, and reconstructed on demand as transformations are applied.

1.2 Heap SSA form

As shown in Figure 2, R2CAT supports Heap SSA-based sparse program analysis. Heap SSA is an extension of Array SSA [4] that generalizes it to support all forms of heap read/write operations, e.g. structure accesses, array accesses, and pointer dereferences.

Heap SSA represents the heap space as an array, introducing a *dphi* function to trace each heap write operation and building the def/use chains for all heap read/write operations. The Heap SSA construction algorithm extends the scalar SSA [2] construction process. For more detail, see [4, 3] and the overview in Section 2.

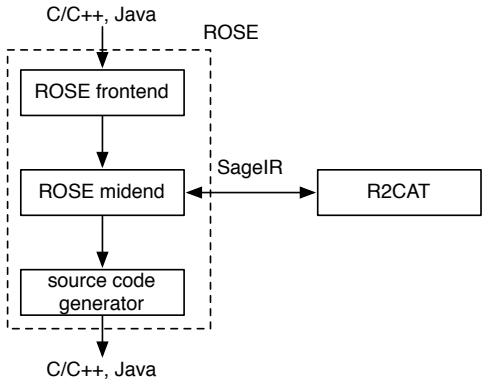

Fig. 1: Interaction between ROSE and R2CAT.

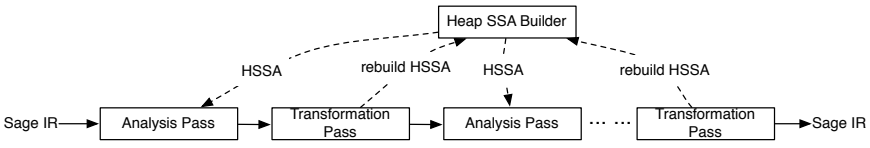

Fig. 2: Internal workflow of R2CAT.

1.3 Analysis

In R2CAT, the analysis passes perform dataflow analysis to identify the relevant values for both scalar variables and heap cells (e.g., array elements, dereferenced pointers). These analyses take advantage of the Heap SSA representation and perform a sparse analysis that reduces the time and space complexity of the analysis process. Section 3 gives more detail about the relevant algorithms.

1.4 LLNL Compositional Framework

LLNL is building a generic compositional dataflow analysis framework within ROSE. One goal is to incorporate R2CAT into the LLNL analysis framework, which is presented in Figure 3. It includes a dataflow analysis driver that maintains a chain of analyses; a common query interface; a series of data structures to facilitate the input/output of analyses. Each *Part* maintains a set of *Abstract Object*s (see details in Section 4.3). An Abstract Object is a high level abstraction of the targets of dataflow analysis, and can have one of two types:

- Abstract value object, e.g. the constant value of a given scalar variable or array element;
- Abstract memory location, which represents scalar variables, array elements, pointers, and label aggregations.

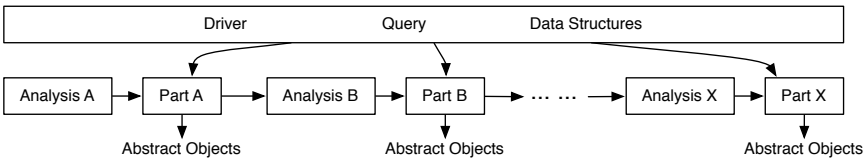

Fig. 3: LLNL Compositional Dataflow Framework.

By mapping the Sage IR (e.g. variable or expression) to its corresponding Abstract Objects, client analyses can query the relevant value for any expression. For example, constant propagation analysis can output a map of expressions and their corresponding constant values. A successor analysis can query the constant value by giving the expression as input. In Section 4, we introduce a detailed account of the LLNL compositional framework's features and discuss how we have adopted the framework for R2CAT.

2 Heap SSA

As mentioned in the previous section, the foundation of R2CAT is the Heap SSA representation. This section describes Heap SSS and how it works to facilitate analyses.

2.1 Running Example

To demonstrate the effectiveness of Heap SSA form, we employ a running example to show how it enables constant analysis to identify constant values for heap cells. In Figure 4, the C code has multiple pointer-dereferenced read/write operations through pointer variables *py3*, *pd1* and *py1*. Scalar SSA form is unable to capture the data flow relationships among reads and writes of the dereferenced pointers, while Heap SSA can.

Figure 5 shows the transformed Heap SSA code that adds *dphi* functions and renamed "heap arrays". In this example, we assume that all memory is represented by a single heap array variable called *MEM*. A pointer value is modeled as an index in the *MEM* array, and each pointer dereference represents a read/write access of an element of the *MEM* array. After Heap SSA naming, we have a set of renamed *MEM* arrays of the form MEM_i, such that every static write of a dereferenced pointer is performed on a distinct heap array thereby satisfying the static single assignment property. Heap SSA form encodes heap-based def/use chains that enable program analyses to identify all heap writes that could reach a given heap read operation. For example, the heap read *MEM2*[*pyi*] in line S7 has a possible heap write at *MEM1*[*py3*] at line S1.

NOTE: an important design decision when implementing program analyses is to determine how many heap arrays to use and what the relationships between heap arrays should be. For example, when analyzing Java programs, it is possible to assign a separate heap array for instances of a specific field since it is not possible for two distinct fields to be aliased with each other in Java. The situation can be more complicated in C due to its pointer aliasing semantics. While the use of a single *MEM* heap array

```
     int y3;
     int * py3 := &y3;
     int d1;
     int * pd1 := &d1;
     int * pyi;
S1:  ... // assigning value to pyi
S2:  *py3 := 99;
S3:  if  C then
S4:    * pd1 = * py3 * 2;
     else
S5:    * pd1 = * pyi * 2;
     endif
S6:  z :=  *pd1;
```

Fig. 4: Example code

will always yield sound and current analyses, the creation of additional heap arrays and *shadow* variables for data accesses can lead to increased precision. For example, a separate heap array, *STACK*, can be created for all local variables (stack locations), and a separate array, *MALLOC* can be created for all data allocated via malloc() calls.

The idea behind shadow variables is to model a single read/write operation in the original program as reads/writes of multiple heap arrays. For example, if a read of `y3` was added to Figure 5, it can be modeled as a direct read of the scalar *y3* as well as a read of *MEM*$[y3]$. Accesses to *STACK* and *MALLOC* heap arrays can also be added as shadow variables.

2.2 Array SSA form

The idea of using the use/def information for heap variables to trace memory accesses was based on Array SSA form [4], which targeted array based programs, i.e., only supported array and index information. Array SSA introduced the *dphi* function that annotates every heap write operation.

In [3], Fink, Knobe and Sarkar introduced the extended Array SSA from, including two extensions:

1. introduce *uphi* functions that are used to trace the heap read operations;
2. model field accesses as accesses to heap arrays in strongly typed programming languages such as Java.

2.3 Heap SSA form

In R2CAT, we generalize the Heap SSA form introduced in for Java to weakly typed programming languages such as C. A global heap variable is introduced (i.e. *MEM* in Figure 5) to represent the single heap array that covers all heap cells, including scalar, array, pointer, label aggregations (i.e. struct/class field accesses). At this level, the *MEM* array includes both stack and heap locations.

```
      int y3;
      int * py3 := &y3;
      int d1;
      int * pd1 := &d1;
      int * pyi;
S1:   ... // assigning value to pyi
S2:   MEM1[py3] := 99
S3:   MEM2:=dphi(MEM1,MEM0)
S4:   if C then
S5:     MEM3[pd1] := MEM2[py3] * 2
S6:     MEM4 := dphi(MEM3, MEM2)
      else
S7:     MEM5[pd1] := MEM2[pyi] * 2
S8:     MEM6 := dphi(MEM5, MEM2)
      endif
S9:   MEM7 := phi(MEM4,MEM6)
S10:  z := MEM7[pd1]
```

Fig. 5: Heap SSA code

The construction of Heap SSA form involves a simple extension to the procedure for building scalar SSA form, and can be summarized as follows for a given C function:

1. Create a heap variable *MEM*;
2. For each heap read/write operation, add pseudo read/write for *MEM*;
3. For each heap write operation, add *dphi* function: *MEM* = *dphi*(*MEM*);
4. Invoke scalar SSA construction by considering the read/write operations for heap variables and *dphi* functions; i.e., the *dphi* function is involved in the renaming process: $MEM_{i+1} = dphi(MEM_i)$.

As with scalar SSA form, the implementation of Heap SSA form is represented as look-aside information in the ROSE IR system; i.e., each Sage IR node (e.g. variable node, or expression node such as array access or struct access) is mapped to its corresponding memory objects and these memory objects are connected by the use/def chain that is the composition of all of the *dphi* functions and their uses.

In the current implementation, we introduce a single heap variable *MEM* that guarantees correctness for weakly typed languages (i.e. C/C++). For a more precise analysis (as mentioned earlier), we will introduce additional heap variables that distinguish distinct heap locations, such as array regions. We will also introduce variables to represent other kinds of variables, such as local variables and stack variables.

3 Analyses

This section discusses the current state of R2CAT, including the analyses implemented, and how they are composed.

3.1 Algorithm Selection

In its current stage, R2CAT includes three program analysis passes:

1. Value numbering;
2. Pointer analysis;
3. Constant analysis.

The corresponding algorithms are selected from well-known academic papers.

Value Numbering The implementation of the value numbering pass is based on the Alpern-Wegman-Zadeck [1] SSA-based value numbering algorithm, which restricts its attention to scalar SSA form. However, the output of this analysis can help improve the precision of constant analysis of both scalars and heap arrays.

Pointer Analysis The pointer analysis implemented in R2CAT is based on the algorithm introduced by Lhotak and Chung [6]. This pointer analysis leverages the advantages of scalar SSA form in supporting strong updates and providing precise flow-sensitive points-to information. For the ”may point to” variables definition (i.e. multiple potential pointer values), this algorithm uses the traditional flow-insensitive approach, i.e. providing all possible pointer values. The output of this analysis can also help improve the precision of constant analysis of heap arrays.

Constant Analysis The constant analysis is based on Wegman & Zadeck's [8] sparse condition constant analysis, applied to Heap SSA form [5]. We extended it to leverage Heap SSA and support constant analysis for heap cells (e.g. array elements, struct fields and dereferenced pointers).

3.2 Interaction between Analyses

All of these analyses interact with each other, i.e., one analysis can provide information to another and improve its precision. The constant analysis for two given heap cells (e.g. pointers or array elements) needs the value numbering or pointer analysis to identify the *must equal* and *must not equal* relations. In the other direction, the constant analysis can benefit both value numbering and pointer analysis by providing precise constant values or branch condition elimination.

Because of these interactions, the analyses are performed in iterative fashion, as shown in Figure 6.

3.3 Query Interface

All of these analyses provide query interfaces to the client, which can be either be a transformation pass or a pass by a different analysis. The query interface takes two input parameters: one is a Sage function node that denotes the current function for query; another is a Sage Expression node that denotes the expression for querying its corresponding lattice.

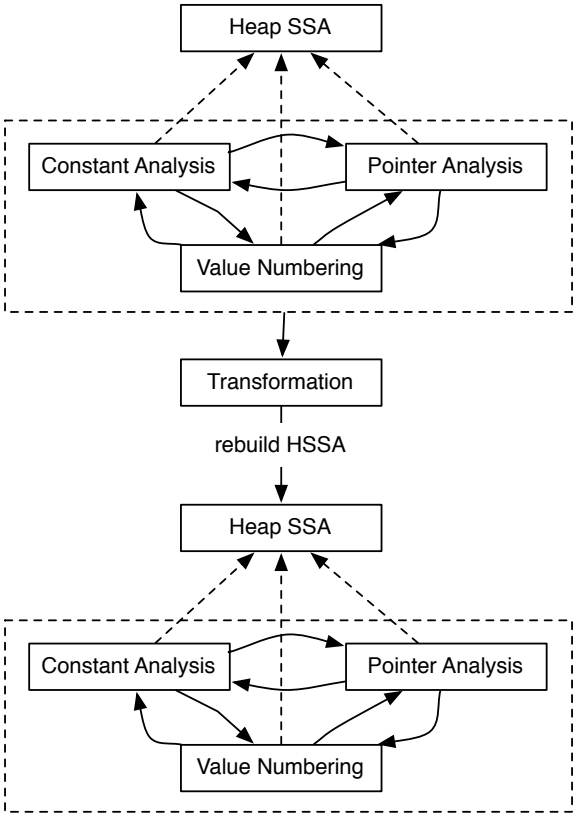

Fig. 6: The composition of constant analysis, value numbering and pointer analysis in R2CAT.

3.4 Composition

All three analyses can provide analysis output to clients. They can be composed into a chain of passes (i.e. analyses or transformations passes) in arbitrary order, and each pass can feed its output to its successor by providing the query interface.

As discussed in section 3.2, these analyses can also be applied iteratively for improving the precision of analysis. Figure 6 presents an example workflow that demonstrates how current analyses could work together within R2CAT. The three analyses are combined into an analysis phase that invokes constant analysis, value numbering and pointer analysis multiple times before the transformation pass.

4 Rice Compositional Analysis

The Rice compositional analysis framework is an extension of the LLNL compositional analysis. The basic aim is to adapt R2CAT to the LLNL compositional analysis framework as follows:

- Driver mechanism and interfaces;
- Support same data structures, i.e. abstract objects;
- Support input/output representation, i.e. lattice interface and expression to value mapping;
- Query interface.

This section discusses the issues listed above.

4.1 LLNL Compositional Analysis Chain

As presented in Figure 3, the LLNL compositional analysis framework has a structure similar to R2CAT, which uses a chain-based compositional structure to drive all analyses and transformations. To support the LLNL framework, we need to support the following C++ interfaces for each analysis:

- *ComposedAnalysis* interface: interacts with the compositional chain;
- *AbstractObject* interface: maintains both value-based objects and memory locations.

4.2 Composed Analysis Interface

ComposedAnalysis is the base class of the analyses that will work off the LLNL compositional analysis chain (see Figure 7). It introduces the functions for maintaining the expressions' lattices (e.g. `Expr2Val` function), the driver mechanism for intraprocedural analysis (e.g. `runAnalysis` function and `visit` function), the context mapping of function invocations (e.g. `genInitState` function and `transferFunctionCall` function), and the interaction between the analysis and composer (e.g. `getComposer` and `setComposer` functions).

Each analysis that implements *ComposedAnalysis* maintains a worklist-based control flow graph (CFG) traversal engine which performs either a forward or backward traversal. An analysis must customize its own transfer function for lattice states. The transfer function is applied at each control flow graph node.

In R2CAT, we are using the Heap SSA form-based forward traversal. It introduces the same interface as the high level LLNL framework (i.e. exposing the `runAnalysis` interface to the composer chain), but uses different internal implementations for analyzing the given input. The major difference is in the control flow graph traversals: the LLNL framework uses a dense CFG traversal, while R2CAT uses a sparse traversal based on def/use chains.

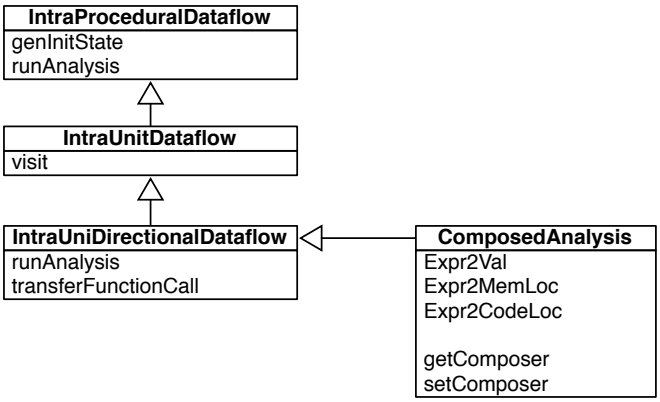

Fig. 7: The class hierarchy for composed analysis.

Interaction between Analyses In the LLNL compositional analysis framework, lattices are used to store analysis results. Each analysis provides the following query interfaces (the class hierarchy for `ComposedAnalysis` is shown in Figure 7):

- `Expr2Val`: returns the value object for a given expression;
- `Expr2MemLoc`: returns the memory location for a given expression;
- `Expr2CodeLoc`: returns the code location (in source program) for a given expression [1].

Both value object and memory location are represented as an abstract object (see details in 4.3). Besides composing analysis results (see Figure 3), the LLNL framework also provides standard lattice representations: finite lattice, infinite lattice and product lattice.

There are three basic types of lattice in the LLNL framework: 1.) a finite lattice, which represents the values corresponding to an expression or any given Sage node (e.g. the constant values in constant analysis); 2.) an infinite lattice, which represents the infinite possible set of values corresponding to an expression; 3.) a product lattice, which represents all expressions' lattice values for a given program point. For adoption of the LLNL framework, we need to use these lattice classes to represent analysis output. The value lattice is the subclass of *ValueObject*, i.e. it has to implement the *mayEqual* and *mustEqual* functions (see Figure 8(a), which shows the lattice used in LLNL's constant propergation analysis, and see more detail in Section 4.3).

In R2CAT, we apply the same implementation, i.e. the lattice implements both the finite lattice interface and the *ValueObject* interface (see Figure 8(b) that shows an example for the lattice used in constant analysis).

[1] We do not handle the code location in the current Rice analysis framework.

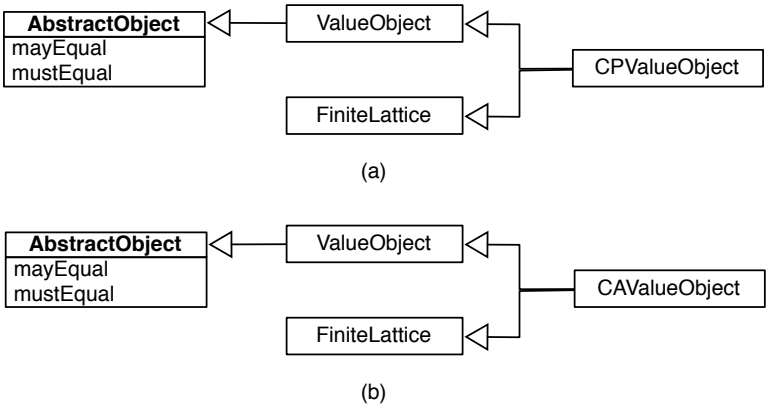

Fig. 8: The class hierarchy for implementing an value lattice.

Intra / Inter Procedural Analysis Each composed analysis in the compositional chain is an intraprocedrual analysis, which is applied to a given input call graph. The LLNL compositional analysis framework implements context insensitive interprocedural analysis in its dirver (see Figure 3), the driver invokes each analysis in the composer chain through the call graph and applies these operations for each function in the call graph:

1. Mapping the initial states from the caller's lattices;
2. intraprocedural analysis: invoking the composed analysis;
3. mapping the lattices back to the caller.

The 1st and 3rd 'mapping' operations are controlled by the Sage IR map shared between caller and callee. Since each IR node has its corresponding lattice, the lattice mapping is based on an IR node mapping. The lattice objects involved in 'mapping' operations are product lattice objects that implement the function `remapML` (see Figure 9), which performs the value mapping. The analysis writer can customize the mapping process by subclassing the product lattice class and reimplementing the `remapML` function.

In R2CAT, we create new product lattices for the analyses (see Section 3): `PTProductLattice`, which is for pointer analysis and performs points-to graph updates between the caller and callee; `VNProductLattice`, which is for value numbering analysis and remaps the value number between the caller and callee; and `CAProductLattice`, which is for constant analysis, passes the constant values between caller and callee. The class hierarchy is shown in Figure 9.

4.3 Abstract Object Interface

The key idea of using the abstract object interface (the `AbstractObject` class in Figure 10) is to represent both value objects and memory location objects with the same super class, since they share *mayEqual* and *mustEqual* functions. Analyses can

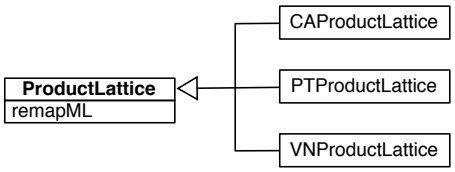

Fig. 9: The class hierarchy for the R2CAT product lattice.

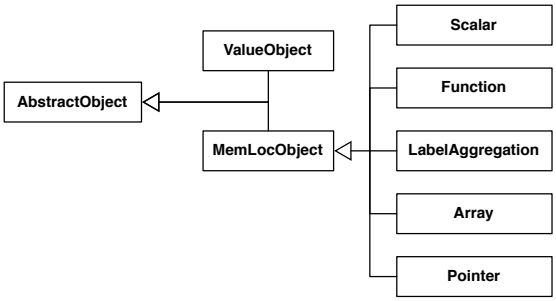

Fig. 10: The class hierarchy if LLNL abstract objects.

customize their own abstract objects to represent different kinds of variables and expressions (e.g. scalar variable, array element), which require different implementations of *mayEqual* and *mustEqual* functions.

Figure 10 presents the class hierarchy of the abstract objects defined in the LLNL compositional dataflow framework. The root class is *AbstractObject*, which has two subclasses: *ValueObject* is used to represent the value, e.g. a given variable's value; *MemLocObject* is used to represent a memory location, e.g. an array reference, or a pointer. Five subclasses of *MemLocObject* for represent the five typical categories of memory locations:

- *Scalar*: scalar variables;
- *Function*: function pointers;
- *LabeledAggregation*: the struct / class fields;
- *Array*: arrays;
- *Pointer*: pointers

These five types of memory location object have different definitions of *mayEqual*/*mustEqual* functions, which must be implemented separately.

Figure 11 gives the class hierarchy of the R2CAT extension of abstract object classes. The *SSAMemLobObj* directly subclasses from *MemLocObject* and maintains a reference to Heap SSA for access to look-aside information (see Section 2). It also introduces a basic implementation of *mayEqual* and *mustEqual* functions by checking

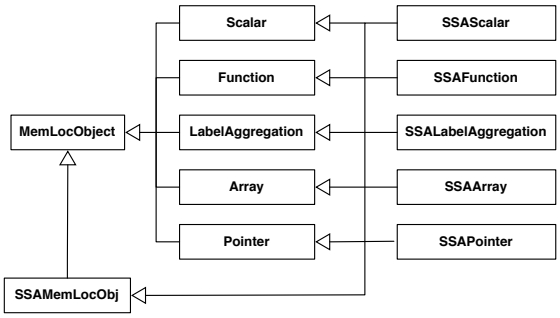

Fig. 11: The class hierarchy of R2CAT's SSA based abstract objects.

the given variable's reaching definition in SSA form. R2CAT also maintains five types of memory locations that are directly subclassed from LLNL memory locations: *SSAScalar*, *SSAFunction*, *SSALabeledAggregation*, *SSAArray* and *SSAPointer*.These memory location objects customize their own *mayEqual* and *mustEqual* functions.

References

1. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *POPL*, pages 1–11, 1988.
2. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.
3. Stephen J. Fink, Kathleen Knobe, and Vivek Sarkar. Unified analysis of array and object references in strongly typed languages. In *SAS*, pages 155–174, 2000.
4. Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In *POPL*, pages 107–120, 1998.
5. Kathleen Knobe and Vivek Sarkar. Conditional constant propagation of scalar and array references using array SSA form. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 33–56. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.
6. Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *POPL*, pages 3–16, 2011.
7. LLNL. LLNL ROSE Compiler Infrastructure. http://http://rosecompiler.org/.
8. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, 1991.